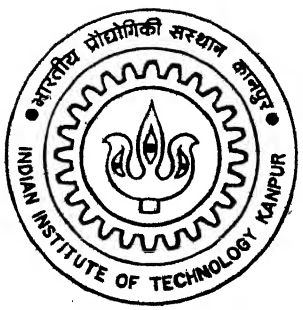
OPTIMIZING LOOPS FOR EXECUTION ON VECTOR PROCESSORS

by

GOVINDARAJ S



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JANUARY, 1995

Optimizing Loops For Execution On Vector Processors

A Thesis Submitted

in Partial Fulfilment of the Requirements

for the Degree of

MASTER OF TECHNOLOGY

by

Govindaraj S

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

January, 1995

CSE-1995-M-GOV-OPT

Certificate

This is to certify that the work contained in the thesis titled **Optimizing Loops For Execution On Vector Processors** by **Govindaraj S (Roll No: 9311106)**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Sanjeev Kumar Aggarwal
Associate Professor,
Department of Computer
Science and Engineering
IIT, Kanpur

January, 1995

. . . to
the memory of my father

Acknowledgements

I would like to express sincere thanks to my thesis supervisor Dr. Sanjeev Kumar Aggarwal for his expert guidance. It was mainly because of his constant encouragement and help that I was able to complete this project on time.

I am extremely grateful to my mother, sister and all my relatives back home for letting me use the opportunities that came my way and being sources of encouragement and support throughout.

I enjoyed my stay at IIT/K thanks to the entire mtech93 junta. I can never forget the "gaali" times I had in the company of pavan, ramkumar, shankar, vineet and barot. Special thanks to muralidhar for being wonderful company and for his suggestions and frank talk-sessions with me. I thank harsha, girisha, madhu, kamat, harish and all other Kannada Sanghaites for their company and help. The daily evening walks with harsha, girisha and madhu provided a break from the daytime monotonicity.

Last but not the least, I would like to thank Shantakka and Prof. Raghavendra for the help, support and affection they showered on me during my stay here.

Abstract

Numerical programs spend more time executing in loops than in the remaining code segments. Consequently, restructuring loops to utilize the underlying architecture should lead to overall reduction in the program execution time. For vector processors, these loop optimizations generally include all transformations that result in generation of better vector code. In this thesis, we shall discuss issues related to the loop optimization techniques like **loop interchange, node splitting,** and **conversion of conditionals** for execution on vector processors. Loop interchange involves interchanging loop positions in a tightly nested loop body, to alter the execution sequence of statement instances in the loop body. Such a transformation moves recurrences involving a majority of statements outwards, thereby allowing the inner loops to be vectorized. Node Splitting entails eliminating dependence cycles having an antidependence edge as a component. This is done by splitting the statement causing the antidependence, with the introduction of a new compiler temporary variable. Finally, conversion of conditionals to vector form involves transformation of IF constructs that do not branch out of the enclosing loop body to equivalent vector WHERE constructs of Fortran 90.

Contents

1 Prologue **1**

1.1 The F90 project : A perspective 2

1.2 Vector Features of Fortran 90 4

1.3 Work done here 5

1.4 Organization of the report. 6

2 Introduction **7**

2.1 Overview of the Vectorizer 7

2.2 The Intermediate Representation 10

2.2.1 The Program Dependence Graph : Concepts 11

2.2.2 Control Dependence Subgraph 12

2.2.3 Data Dependence Subgraph 14

2.2.4 An Example 14

3 Data Dependence Analysis **16**

3.1 Definitions, Concepts and Terminology 16

3.2 Dependence Testing 22

3.2.1 GCD Test 23

3.2.2 Extended GCD Test 23

3.2.3 Banerjee's Test 24

3.2.4 Conclusions 29

3.2.5 Implementation Notes 29

4 Code Restructuring **30**

4.1 Loop Interchange 30

4.1.1 Interchange Prevention 32

4.1.2 Profitability of interchange 38

4.1.3 Interchange effects on dependences. 38

4.1.4 Two approaches to interchanging loops 39

4.2 Node Splitting 41

4.3 Vector Code Generation 43

4.4 Vectorizing Conditionals 45

5 Epilogue **50**

5.1 Summary 50

5.2 Status of Implementation 51

5.3 Testing of the software 51

5.4 Future Directions 52

List of Figures

1.1 The F90 Compiler 3

2.1 The F90 Vectorizer 8
2.2 The Control flow graph and PDG for the example program 15

3.1 True dependence between two statements 17
3.2 Antidependence between two statements 17
3.3 Output dependence between two statements 18
3.4 A general loop nest for linear dependence problem 18
3.5 Rectangular iteration space 26
3.6 Trapezoidal iteration space 26
3.7 Algorithm for finding bounds in trapezoids 28

4.1 Dependence patterns in loops 34
4.2 Dependence patterns among a sequence of statements 35
4.3 Algorithm for interchange of adjacent loops 40
4.4 Algorithm for general loop interchange 42
4.5 Sample program for Node Splitting 43
4.6 Node S_1 has been split 43
4.7 Node Splitting followed by Reordering 44
4.8 Algorithm for Vector code generation 45

Chapter 1

Prologue

Computers have been applied to problems from various fields. As a result, millions of lines of serial code have accumulated over the years into what are popularly known as *dusty decks*. Technology too, has progressed to meet this demand for computation power in the form of new architectural paradigms. One such advancement is the vector computer, which exploits concurrency in data to speed up computations. Just as there are "optimizing" compilers for serial computers, which generate code that can make "optimal" usage of the underlying serial architecture, compilers that do the same for vector processors are also needed. Extending the concept a little further, it is possible to develop a compiler that takes a sequential program, performs various optimizations on it, and produces code that effectively exploits the underlying vector architecture. This approach serves two purposes:

1. Absolves the user from the burden of knowing the intricacies of underlying architecture (for writing programs that execute faster on that machine).

2. Already written sequential programs can be executed without being rewritten for the vector architecture.

The Fortran 90 (F90) [11] vectorizing compiler being built at the Indian Institute of Technology, Kanpur, is one such effort directed towards meeting the above stated purposes.

1.1 The F90 project : A perspective

The F90 vectorizing compiler being built at IIT/K envisages producing an 'optimized' and 'vectorized' Fortran 90 program from an input program from the Fortran 77–Fortran 90 class of languages. Figure 1.1 depicts the overall structure of the F90 compiler [20, 21]. The functions of each of these phases, very briefly are as follows:

- The **preprocessor** handles compiler directives such as file inclusion and presents to the lexical analyzer an "expanded" version of the input program.
- The **lexical analyzer** accepts the input stream of characters from the preprocessor, identifies the lexemes and feeds to the parser.
- The **parser** uses the lexemes received from the lexical analyzer and recognizes the syntax structures formed by them, simultaneously constructing the syntax tree representation of the input program.
- The **optimizer** performs *scalar* code improvement transformations such as constant propagation and folding on the input syntax tree after analyzing the program for aliases and reaching definitions.
- The **vectorizer** performs *vector* optimizations on the program to uncover latent parallelism in the program. These optimizations include detecting auxiliary induction variables in loops, loop restructuring and converting array references and conditionals to equivalent vector forms, if possible.
- The **backend**, by a process called *unparsing*, generates the scalar and vector optimized Fortran 90 equivalent of the input program. This is in aid of the user, who can use it to analyze the optimizations performed on the input program. We also hope that the user will use this feedback to issue the compiler directives to aid optimization.
- The **translator** phase converts the syntax tree output by the vectorizer to a low level internal representation (*operator tree*) in which each non-leaf node represents an operator and leaf node, an operand.

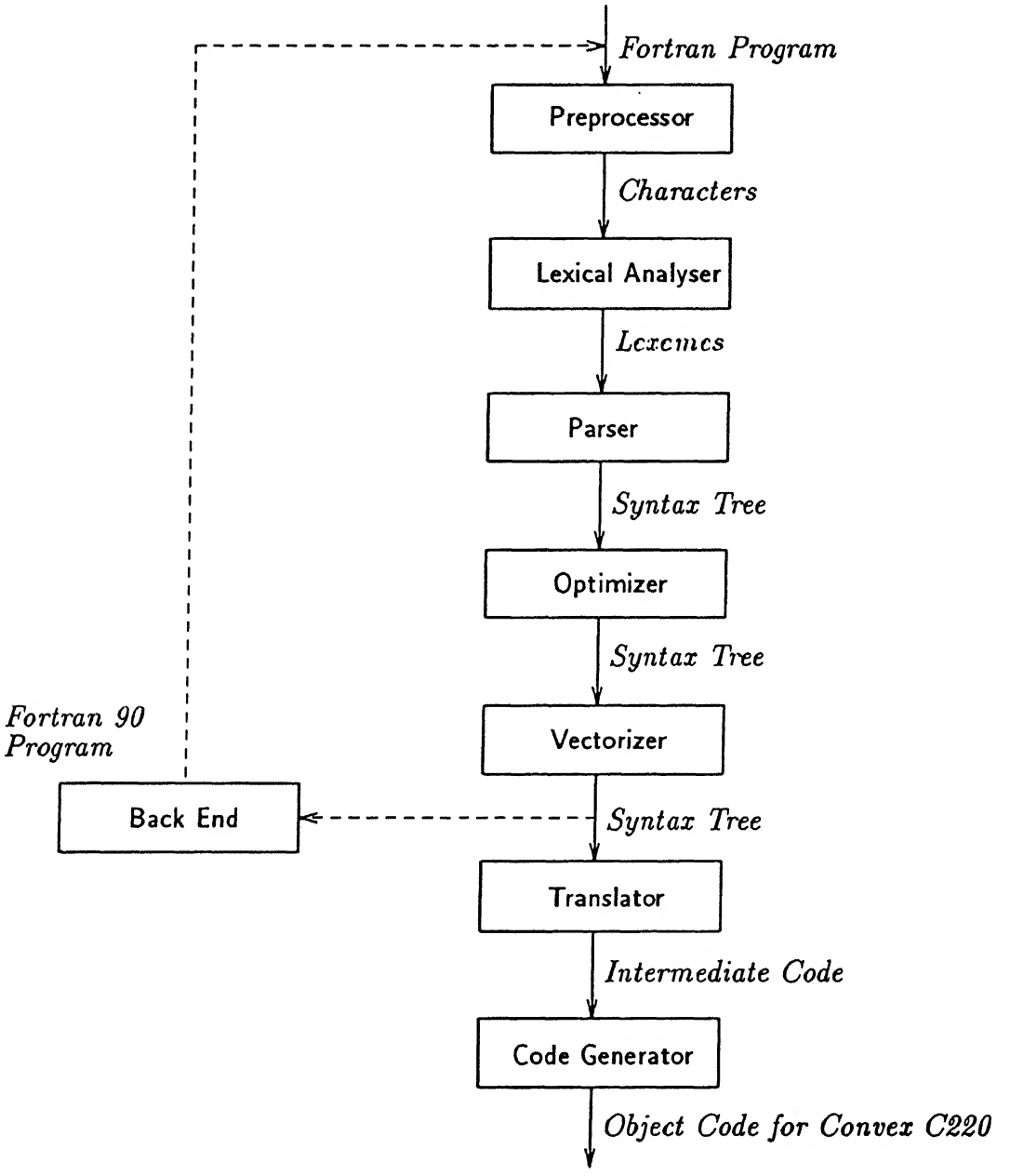


Figure 1.1: The F90 Compiler

- The **code generator** generates the assembly language equivalent of the operator tree input it receives from the translator.

1.2 Vector Features of Fortran 90

Fortran 90 [11] provides vector constructs for array references and conditional assignments. These advanced features can effectively utilize the underlying vector hardware. Here, we give a brief description of these features:

Array assignment. If A and B are two arrays of the same dimension N, then

A = B

assigns B(i) to A(i) for each $1 \leq i \leq N$.

Sections of arrays can also be referenced using the triplet notation. If M $\leq$ N, and A and B are two N $\times$ N arrays, then

A(1:M:1, I) = B(J, 1:M:1)

assigns the elements of the J^{th} row of B to the I^{th} column of A.

In this triplet notation, the first two components specify the range of iteration and the third component specifies the "stride" for the index vector in that subscript position.

Conditional Assignments. Fortran 90 provides a facility for making conditional assignments through WHERE statements or constructs. We will only present an instance of its usage here. The syntax is presented later (Chapter 4) when we talk about vectorizing conditionals.

If A and B are arrays of same dimension N, then

```
WHERE ( A .NE. 0 ) B = B / A
ELSEWHERE B = 1
```

makes a test for A(i) $\neq$ to 0 and makes the assignment to B(i), $1 \leq i \leq N$, accordingly. Here A is called he *mask* array.

1.3 Work done here

The work outlined in this report concerns the **vectorizer** phase of the F90 compiler. In particular, we will be concentrating on the following *code restructuring* techniques:

Loop Interchange. This is one of the techniques under the *Reordering transformations* category of restructuring techniques, which includes all transformations that affect the order in which the code is executed. Loop interchange is a powerful aid for vectorization. It alters the order in which the loops inside a loop nest are executed by interchanging positions of the loops. The data dependences within the loop nest have to be analyzed to safely perform this operation. Later, we formalize the notion of *interchange preventing dependences,* and provide algorithms for determining when a loop interchange is valid and profitable.

Breaking cyclic dependences. Presence of some "pseudo" dependences such as *antidependences* as components of dependence cycles inhibits vectorization. Such dependence cycles can be broken by introducing new compiler variables for the antidependence causing variables. This technique, called *node splitting* is described in detail Chapter 4.

Vectorizing conditionals. The theory of *restructuring compilers* is oriented towards data dependences. The fact that most vector computers provide some facility for simultaneous conditional execution of statements adds a whole new

dimension to the proceedings. This is so, because IF constructs introduce what are known as *control dependences* which must be taken into account while vectorizing. In Chapter 4, we do an in-depth analysis of the various issues involved in vectorizing IFs.

1.4 Organization of the report.

The rest of the report is organized as follows

- **Chapter 2** overviews the vectorizer and the intermediate representation used, the *Program Dependence Graph*.
- **Chapter 3** deals with the concepts of *data dependence and analysis.*
- **Chapter 4** details various *restructuring techniques* applicable to vectorization
- **Chapter 5** summarizes the work done, suggests extensions to the project and outlines the testing carried out.

With this *prologue* we proceed to take a closer look at the various issues.

Chapter 2

Introduction

2.1 Overview of the Vectorizer

This section presents an overview of the different phases of the vectorizer which is depicted in Figure 2.1. The functions of the various stages of the vectorizer are very briefly outlined here. Most of these phases are explained in detail in this and the following chapters.

Control Flow Graph generation: The scalar optimizer, the stage preceding the vectorizer, produces syntax tree, which is input to the vectorizer. The structure of this syntax tree is similar to that of the syntax tree produced by the parser. This stage of the vectorizer concerns itself with the conversion of this syntax tree into a control flow graph.

Control Dependence Subgraph generation: This is the first stage in the construction of the Program Dependence Graph (PDG) [10]. The control dependence subgraph of the PDG is built from the control flow graph using the method explained in Section 2.2.

Data Dependence Subgraph generation: Scalar dependence edges are added to the control dependence subgraph using classical data flow analysis techniques [2]. The output of this phase is the complete PDG.

Loop Normalization: This phase takes the PDG, and transforms loops so that the loop induction variables iterate from 1 to some upper bound by

increments of 1. This is accomplished sometimes by the introduction of new induction variables. All references to the old induction variables within the loops are replaced by expressions in the new induction variables.

Auxiliary induction variable elimination: Sometimes, the subscript expressions of the array references within loops will be in terms of some 'auxiliary' variables that go in tandem with the actual loop induction variable. These auxiliary induction variables can be eliminated by replacing them with references to the actual loop induction variables.

With the completion of this phase, we will have all the loop induction variables incrementing from 1 to some upper bound in steps of 1, and all array subscripts converted to linear functions of the loop induction variables. Such loops are said to be *standardized.*

Data dependence analysis: Scalar data dependence analysis treats any reference to an array element to be a reference to the whole array. Such conservative estimation of data dependences will not suffice for effective exploitation of parallelism inherent in programs. More accurate data dependence information can be got by analyzing the array subscripts. This phase adds the array data dependence edges to the PDG.

Loop Interchange: In many cases, it is possible to enhance vectorization by interchanging the nesting order of loops. Called *loop interchange,* this transformation is semantically feasible and profitable under certain conditions. This phase concerns itself with identifying and performing feasible and profitable loop interchanges.

Node Splitting: Presence of some *pseudo* dependences in a dependence cycle hampers vectorization. With some ingenuity, such dependence cycles can be broken. Node Splitting is one such technique, which in the literal sense means splitting a node (statement) into two nodes (statements), in the process introducing a new variable to eliminate the *pseudo* dependence. This phase entails breaking such dependence cycles.

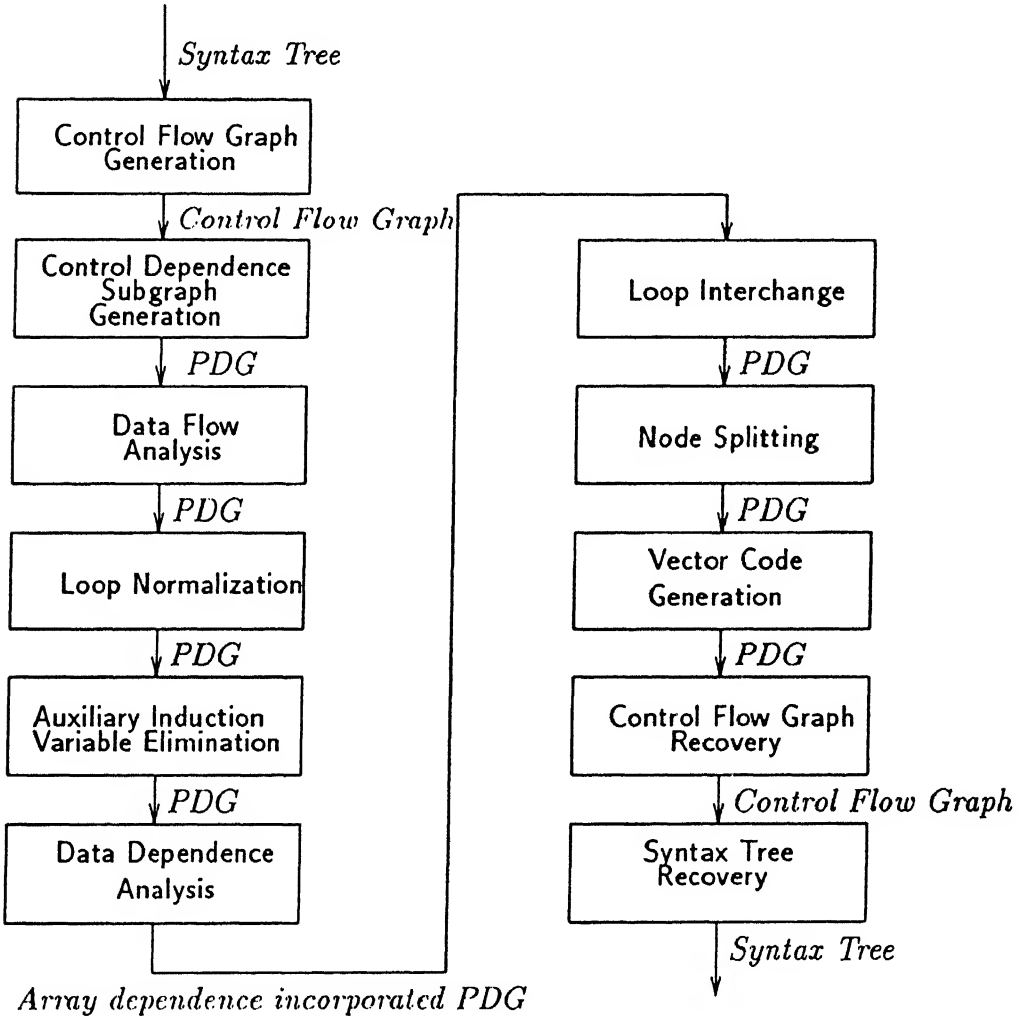


Figure 2.1: The F90 Vectorizer

Vector code generation: This phase concerns itself with identifying vectorizable statements and generating the vector code for them. Wherever possible, vector constructs of Fortran 90 replace array references and IF constructs in the vectorized version of the input program.

Control flow graph recovery: This phase reconstructs the control flow graph from the PDG. The control flow graph generated, if different from the one that the vectorizer received as input, will represent the vectorized version of the input program.

Syntax tree recovery: This phase ensures inter-operability of the vectorizer with the other phases of the F90 compiler; it generates the syntax tree acceptable by the next phase (Translator), the structure of which is the same as the syntax tree generated by the parser. The Translator is non-existent as of now. Hence the output of this phase goes to the optional *unparser* phase of the compiler which in effect generates the Fortran 90 equivalent of the syntax tree.

2.2 The Intermediate Representation

The F90 vectorizer uses the *Program Dependence Graph* (PDG) [10] as the intermediate representation. The PDG provides a unified framework in which various optimizations and program restructuring techniques may be applied. Before the PDG was proposed, some optimizing compilers used data dependence graphs [19] for an explicit representation of the definition-use relationships implicitly present in a source program, and the control flow graph [2] to represent the control flow relationships in a program. The control flow graph however, has the undesirable property of a fixed sequencing of operations that need not hold. The PDG explicitly represents both the *essential* data dependences, as present in the data dependence graph, and the *essential* control relationships, without the unnecessary sequencing present in the control flow graph. These dependence relationships determine the *necessary* sequencing between operations, exposing

potential parallelism.

2.2.1 The Program Dependence Graph : Concepts

The PDG represents a program as a directed graph in which

- the nodes are statements and predicate expressions, and
- the edges incident to a node represent
  - the data values on which the node's operations depend, and
  - the control conditions on which the execution of the operations depend.

The set of *all* dependences for a program may be viewed as imposing a partial ordering on the statements and predicates in the program that must be followed to preserve the semantics of the original program.

Dependences result from two separate effects. First, a dependence exists between two statements when they have references to some common memory location, and at least one of them writes that location. Such dependences are the data dependences. We will talk about data dependence in detail in Chapter 3. Second, a dependence exists between a statement and a predicate whose value immediately controls the execution of the statement. For example, in the sequence,

```
S1:   if (P) then
S2:       C = A * B
      endif
```

S_2 depends on predicate P since the value of P determines whether S_2 is executed. Dependences of this type are control dependences.

The PDG unifies these dependences in the sense that it is composed of a control dependence subgraph and a data dependence subgraph of the program being represented.

We will now briefly look at the construction of the PDG beginning with some essential formal definitions [10].

Control flow graph. A control flow graph is a directed graph G augmented with a unique entry node START and a unique exit node STOP such that each node in the graph has *at most* two successors. The nodes with two successors are assumed to be having attributes "T" (true) and "F" (false) associated with the outgoing edges, and for any node N in G, there exists a path from START to N and a path from N to STOP.

Post-dominators. A node V is post-dominated by a node W in G if every directed path from V to STOP (not including V) contains W. Note that, by definition, a node *never* post-dominates itself.

Control Dependence. Let G be a control flow graph. Let X and Y be nodes in G. Y is control dependent on X iff

1. there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y, and
2. X is not post-dominated by Y.

2.2.2 Control Dependence Subgraph

The algorithm below gives the procedure for construction of the control dependence subgraph of the PDG. The reader is, however, encouraged to look up [10] for a detailed discussion on the construction and applications of the PDG.

Step 1: Augment the control flow graph with a special predicate node ENTRY that has one edge labeled "T" going to START and another edge labeled "F" going to STOP. ENTRY corresponds to whatever external condition causes the program to begin execution.

Step 2: Construct a post-dominator tree for the augmented control flow graph. Computing the post-dominators in the control flow graph is equivalent to computing dominators [2] in the reverse control flow graph. The algorithm in [13] can be used for this purpose.

Step 3: Let S consist of all edges (A, B) in the control flow graph such that B is not an ancestor of A in the post-dominator tree (i.e., B does not post-dominate A). Proceed by examining each edge (A, B) in S. If L denotes the least common ancestor of A and B in the post-dominator tree, either L is A or L is the parent of A in the post dominator tree (for a proof of this, see [10]).

Case L = parent of A. Make all nodes in the post-dominator tree on the path from L to B, including B but not L, control dependent on A.

Case L = A. Make all nodes in the post-dominator tree on the path from A to B, including A to B, control dependent on A.

Step 4: This step concerns the addition of *region nodes* (predicate nodes) to summarize the set of control conditions for a node and group all nodes with the same set of control conditions together. First, we consider the set **CD** of control dependence predecessors of each nonregion node that has other than a single unlabeled control dependence predecessor. A region node R is created for **CD** and each node in the graph whose set of control dependence predecessors is **CD** is made to have only the single control dependence predecessor R. Finally, R is given **CD** as its set of control dependence predecessors.

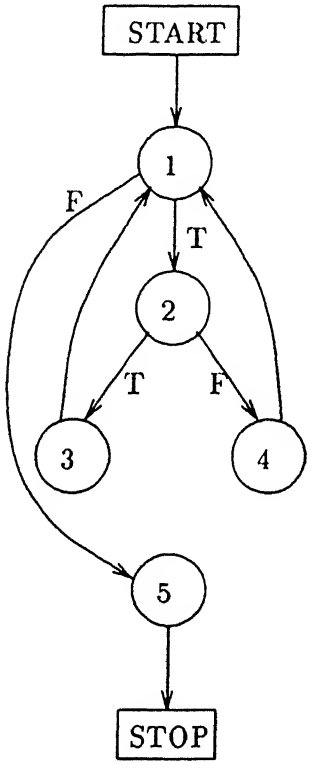
The entire algorithm for the construction of the control dependence subgraph takes time $O(N^2)$, where N is the number of nodes in the control flow graph.

2.2.3 Data Dependence Subgraph

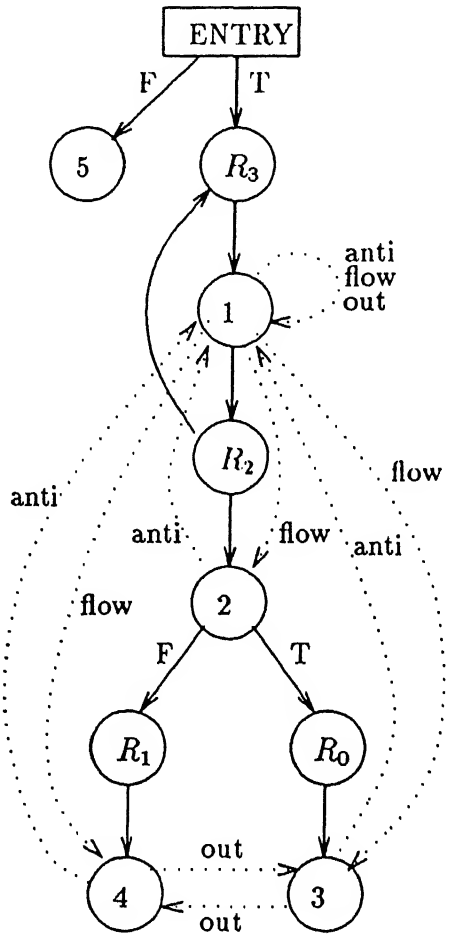
Classical data flow analysis techniques [2] and the array dependence analysis presented in Chapter 3 are used for the construction of the data dependence subgraph of the PDG. Data flow analysis computes the set of reaching definitions [2] for each statement. The results of the data flow computation are then explicitly represented in the PDG by adding edges from the definition nodes to the corresponding use nodes. References to any element of an array are treated as references to the whole array. Depending on the order of the definition and use, edges are marked as *flow, anti* or *output* [19, 2] dependence edges. Data dependence analysis [7] is then used to update the data flow subgraph. Data dependence analysis considers subscripts of the array references for more accurate computation of the flow of data. We will go into the details of data dependence analysis in Chapter 3.

2.2.4 An Example

We consider a small sample program and illustrate the structure of the PDG constructed for it. Figures 2.2(a) and 2.2(b) show the control flow graph and the complete PDG for the program. The solid lines represent control edges while the dotted lines represent data dependences. Control dependences are labelled with "T" and "F" for TRUE and FALSE conditions of control flow, respectively. Unlabelled control edges represent unconditional execution of the successor statement. Data dependences are labelled with "flow", "anti" and "out" which represent the true, anti and output dependences, respectively.



(a)



(b)

Figure 2.2: The Control flow graph and PDG for the example program

S_1: DO I = 1, 100, 1

S_2: IF (X (I) .GT. 0) then

S_3: Y(I) = 1

ELSE

S_4: Y(I) = X(I-1)

END IF

ENDDO

Chapter 3

Data Dependence Analysis

As we mentioned in Chapter 2, dependences in programs occur in two types: control and data dependences. Control dependences occur as a consequence of the *flow of control* in a program. For instance, the statements in the **if-then** block of a program are control dependent upon the **if** test. The other type of dependence, data dependence occurs as a consequence of the *flow of data* in a program. Data dependences have been given thorough treatments in [7, 25]. In what follows, we present the bare minimum of the concept, terminology and analysis of data dependences. This should be sufficient to help the reader appreciate the importance of this topic, which stems from the fact that complete vectorization of statements involved in cyclic dependences is not possible. Effective tests have been developed to identify the presence or absence of such dependence cycles, and the later phases utilize this information to generate vector code.

3.1 Definitions, Concepts and Terminology

Statement T depends on statement S (S Δ T) if there exist an instance S' of S, an instance T' of T, and a memory location M, such that

1. Both S' and T' reference M, and at least one of the references is a write
2. In the serial execution of the program, S' is executed before T', and

S : M = ...
T : ... = M

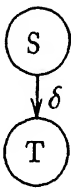


Figure 3.1: True dependence between two statements

3. In the same execution, M is not written between the time S' finishes and the time T' starts.

Kuck and others [19] classify the dependences based upon the two types of references to M [5].

Flow dependence or True Dependence. S' writes and then T' reads it. Flow dependence is denoted by S δ T and is illustrated by Figure 3.1.

Antidependence. S' reads M and then T' writes it. Antidependence is denoted by S $\bar{\delta}$ T and is illustrated by Figure 3.2.

Output Dependence. S' writes M and then T' writes it. Output dependence is denoted by S δ° T and is illustrated by Figure 3.3.

Thus we can say that $\Delta = \delta \cup \bar{\delta} \cup \delta^{\circ}$ [5], which means T depends on S if any of the above three dependences hold between S and T. In all the three cases, S is called the *source* and T the *sink* of the dependence.
Formally, given a loop nest as in Figure 3.4, where m represents the number of subscripts of array X, a dependence from S to T exists *iff* the following holds:

$$f_k(i_1, \ldots, i_n) = g_k(j_1, \ldots, jn), \quad \forall k,\ 1 \leq k \leq m$$

Here i_q, j_q are instances of I_q, $1 \leq q \leq n$.

S : ... = M
T : M = ...

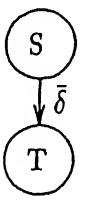


Figure 3.2: Antidependence between two statements

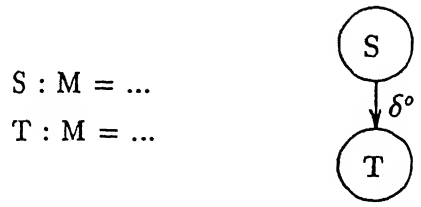


Figure 3.3: Output dependence between two statements

DO $I_1 = 1, N_1$
  Do $I_2 = 1, N_2$
    $\vdots$
    DO $I_n = 1, N_n$
    S: $\quad X(f_1(I_1, \ldots, I_n), \ldots, f_m(I_1, \ldots, I_n)) = \ldots$
    T: $\quad \ldots = X(g_1(I_1, \ldots, I_n), \ldots, g_m(I_1, \ldots, I_n))$
    ENDDO
    $\vdots$
  ENDDO
ENDDO

Figure 3.4: A general loop nest for linear dependence problem

Distance Vectors. For each data dependence involving statements $S(i_1, \ldots, i_n)$ and $T(j_1, \ldots, j_n)$ where n loops surround S and T, we define the r^{th} distance D_r or $D_r(\delta)$ to be $D_r = j_r - i_r$, $(1 \leq r \leq n)$. The n-tuple $D = (D_1, D_2, \ldots, D_n)$ is called dependence *distance vector* [23].

Distance represents the number of iterations that must elapse between the execution of the source and the sink of the dependence, and is important for some advanced compiler optimizations like *shrinking* [23]. Distance also implicitly makes known the direction of the dependence arrow.

Direction Vectors. In many cases it is not possible to compute accurately the distance vector of a dependence. A simpler but less exact description of the data dependence direction is the sign of the dependence [25].

Thus, for each data dependence involving statements $S(i_1, \ldots, i_n)$ and $T(j_1, \ldots, j_n)$ where n loops surround S and T, we define the r^{th} direction d_r or $d_r(\delta)$ to be $d_r = \text{sign}(j_r - i_r), (1 \leq r \leq n)$. Here,

- sign(x) = "<" if x < 0
- sign(x) = "=" if x = 0
- sign(x) = ">" if x > 0.

The n-tuple $d = (d_1, d_2, \ldots, d_n)$ is called dependence *direction vector* [7]. The direction of dependence for a loop is

forward, if $\text{sign}(j_r - i_r)$ is "<", i.e., a dependence exists from an iteration to a subsequent iteration

equal, if $\text{sign}(j_r - i_r)$ is "=" i.e., a dependence exists in the same iteration

backward, if $\text{sign}(j_r - i_r)$ is ">" i.e., a dependence exists from one iteration to an earlier iteration

An asterisk ("*") is used when the direction is unknown or all of "<", ">", "=", apply.

Loop Carried and Loop Independent dependences. Dependences can be further classified as being "carried" by a particular loop or being independent of loop iterations.

Referring to Figure 3.4, we say statement T has a *loop carried dependence* on S (denoted S δ T) if there exists some i_r and j_r such that $1 \leq i_r < j_r \leq N_r$, $(1 \leq r \leq n)$ and each of

$$f_k(i_1, \ldots, i_r, \ldots, i_n) = g_k(j_1, \ldots, j_r, \ldots, j_n),\ 1 \leq k \leq m$$

holds. Note that self dependence is merely a special case of loop carried dependence.

Again referring to Figure 3.4, we say statement T has a *loop independent dependence* on S (denoted Sδ_∞T) if T appears after S in the loops (i.e., T lexically follows S) and there exists some iteration $i_r, 1 \leq i_r \leq N_r$, $(1 \leq r \leq n)$ such that each of

$$f_k(i_1, \ldots, i_r, \ldots, i_n) = g_k(j_1, \ldots, i_r, \ldots, j_n),\ 1 \leq k \leq m$$

holds.

It is important that we identify the "carrier" loop of a dependence [5]. This is so, because many of the advanced loop restructuring optimizations performed later depend on this identification. We will define this more formally.

Let n_1 be the number of loops containing statement S,

$$S:\ X(f(x_1, \ldots, x_{n_1})) = \ldots$$

and n_2 be the number of loops containing statement T,

$$T:\ A = G(X(f(x_1, \ldots, x_{n_2}))$$

Let f and g be subscript mappings

$$f : Z^{n_1} \rightarrow Z^m$$

$$g : Z^{n_2} \rightarrow Z^m$$

where m is the number of subscripts for array X, Z is the set of all integers, $G()$ is an arbitrary function. $x_1, x_2, \ldots$ denote the induction variables for the loops. Further, the upper bound of the i^{th} loop surrounding S is assumed to be M_i; the upper bound of the i^{th} loop surrounding T is assumed to be N_i. So if n is the number of common loops surrounding the two statements, $M_i = N_i$, forall $1 \leq i \leq n$.

We say statement T depends on statement S with respect to *carrier* k [5] ($k \leq n$), written S δ_k T, if there exist $(i_1, \ldots, i_{k-1})$, $(j_{k+1}, j_{k+2}, \ldots, j_{n_1})$, $(l_{k+1}, l_{k+2}, \ldots, l_{n_2})$, and integers ζ_1, ζ_2 in the following regions :

$$1 \leq i_q \leq N_q \ \forall q \ s.t. \ 1 \leq q \ and \ q < k$$

$$1 \leq j_q \leq M_q \ \forall q \ s.t. \ k < q \ and \ q < n_1$$

$$1 \leq l_q \leq N_q \ \forall q \ s.t. \ k < q \ and \ q < n_2$$

$$1 \leq \zeta_1 < \zeta_2 \leq N_q$$

such that the following equation holds:

$$f(i_1, \ldots, i_{k-1}, \zeta_1, j_{k+1}, \ldots, j_{n_1}) = g(i_1, \ldots, i_{k-1}, \zeta_2, l_{k+1}, \ldots, l_{n_2})$$

Depth of a dependence. We say that statement T depends on statement S at *depth d* (denoted S Δ_d T)[5], if there exists a $k \geq d$ such that S δ_k T.

Nesting Level of dependence. For statements S and T, η(S,T), the nesting level of the *direct dependence*[5] of T on S, is defined as the maximum depth at which the dependence exists.

For statements S and T, η(S,T), the nesting level of the *dependence*[5] of T on S, is defined as the maximum depth d at which there exists a *path* $(P_0, P_1, \ldots, P_n)$ such that,

$$S = P_0 \, \Delta_d \, P_1 \, \Delta_d \, P_2 \, \Delta_d \, \ldots \, \Delta_d \, P_n = T$$

Parallelism Index (ρ). Consider a statement S that depends upon itself. The *parallelism index* of S, $\rho(S)$ is defined as

$$\rho(S) = m - \eta(S, S)$$

where m is the number of loops containing S. The implication of the parallelism index is that if $\rho(S) > 0$, then S may be executed in parallel in the innermost $\rho(S)$ loops surrounding it.

3.2 Dependence Testing

A majority of array references in practice have been observed to be linear functions of the loop induction variables, which *somewhat* reduces the complexity of solving the dependence problem. Equations involving linear functions of variables (i.e., equations of degree 1) are commonly know as *linear diophantine equations* and have the form

$$a_1x_1 + a_2x_2 + a_3x_3 + \ldots + a_nx_n = c$$

where $x_1, x_2, \ldots, x_n$ are the variables to be solved.

Further, the loop induction variables take on only integer values and are bound by some upper limits. This fact reduces the problem to deciding whether a solution exists for a given set of linear diophantine equations in a bounded region. In other words, *the linear dependence problem always involves solving a system of linear diophantine equations subject to a set of linear constraints.*

There are two major types of dependence tests: *exact* and *inexact or approximate*. In an exact test, we actually find the general (integer) solution to the system of equations and test to see if a solution exists that fits all the constraints. In an inexact test, we check if there is an integer solution to the system or to each individual equation, subject to the constraints. Inexact tests are much less expensive compared to the exact tests as they address the simpler question "does dependence exist?" rather than the more specific "what iteration instances cause dependence?" and can be tuned to find dependence at a particular level.

3.2.1 GCD Test

This is an inexact test which borrows the concept of *Greatest Common Divisor* from the realm of Number Theory and attempts to apply the same to solve the dependence problem.

Algorithm 3.1: GCD Test [7] Given a system of m linear diophantine equations

$$f_r(x) = a_{r1}x_1 + a_{r2}x_2 + \ldots + a_{rn}x_n = c_r,\ (1 \leq r \leq m)$$

in n variables, then, for each r, $1 \leq r \leq m$, do the following:

Step 1 : Find $gcd(a_{r1}, a_{r2}, \ldots, a_{rn})$ using Euclid's algorithm

Step 2 : If this *gcd* does not divide c_r, the r_{th} equation (and hence the system) has no solution; terminate the algorithm

Step 3 : Assume there is dependence

The GCD test is quick, but is relatively ineffective in practice. The *gcd* of a set of numbers is one more often than not, which reduces the utility of this test.

3.2.2 Extended GCD Test

Algorithm 3.1 tests each linear diophantine equation individually. Instead, if we test for the existence of a solution to the system as a whole, there is some scope for improvement. The resulting test, called *Generalized GCD test* by Banerjee [7] borrows some concepts from Matrix Theory. We will briefly look at some matrix terminology.

Unimodular matrix. A square matrix A is unimodular if

$$determinant(A) = \pm 1$$

Echelon matrix. An $m \times n$ matrix is an echelon matrix if it has the following form:

1. For some k in $0 \leq k \leq m$, the last k rows consist entirely of zeros

2. For $1 \leq i \leq m-k$, if the first nonzero element on row i lies in column j, then each element in column j after row i is zero.
3. The first nonzero elements in the rows are in an echelon form.

Now we will give the *extended GCD test.*

Algorithm 3.2: Extended GCD Test [7] Consider a system of m linear diophantine equations

$$f_r(x) \equiv a_{r1}x_1 + a_{r2}x_2 + \ldots + a_{rn}x_n = c_r, \ (1 \leq r \leq m)$$

in n variables. Let A denote the $n \times m$ coefficient matrix $[a_{rk}]^t$, C the $m \times 1$ matrix $[c_r]^t$. Then,

Step 1: Find U, an $n \times n$ unimodular integer matrix and D an $n \times m$ an echelon integer matrix such that UA = D [7].

Step 2: If an $m \times 1$ integer matrix T satisfying TD = C does not exist, then the system has no solutions and the algorithm terminates.

Step 3: Assume there is a solution.

3.2.3 Banerjee's Test

Banerjee noticed that the *Intermediate Value Theorem* has an important bearing on the dependence problem and applied it to try and solve the same. Here, we produce the theorem from [7].

Theorem 3.1 (Intermediate Value Theorem) *Let f be a continuous real valued function on R^n. Let b, B denote any two values of f on a connected set $\Re \subset R^n$, and suppose that $b \leq c \leq B$. Then the equation*

$$f(x) = c$$

has a solution $x \in \Re$. □

The implication of this theorem is that, if we evaluate the upper and lower bounds of f and verify if c lies within these bounds, then there exists a real solution x in $\Re$ that will satisfy $f(x) = c$. But this is still not conclusive as the existence of a real solution does not guarantee the existence/absence of an integer solution, which is what is sought.

3.2.3.1 Banerjee's inequality

Consider a system of m linear diophantine equations

$$f_r(x) \equiv a_{r1}x_1 + a_{r2}x_2 + \ldots + a_{rn}x_n = c_r,\ (1 \leq r \leq m)$$

in n variables and a nonempty region $\Re \subset Z^n$. If c_r does not satisfy

$$b_{low}(f_r, \Re) \leq c_r \leq b_{up}(f_r, \Re)$$

then there is no solution to the r^{th} equation (and hence to the system).

3.2.3.2 Determining the bounds of $\Re$

The region $\Re \subset R^n$ in which we search for the existence of an integer solution is defined by the iteration space. In general, the region defined will be either *rectangular* (loop bounds remain unchanged once execution of loop nest begins) or *trapezoidal* (loop bounds being functions of other variables).

The examples in the Figure 3.5 and 3.6 illustrate the concept of "bounds" of loops. Banerjee proposed algorithms to find the bounds of $\Re$ in both the cases.

Notation

If a is a real number, the *positive* and *negative* parts of a are defined as

$$a^+ = max\{a, 0\}$$

$$a^- = min\{-a, 0\}$$

```
DO I1 = 1, N1
   DO I2 = 1, N2
      .
      .
   ENDDO
ENDDO
```

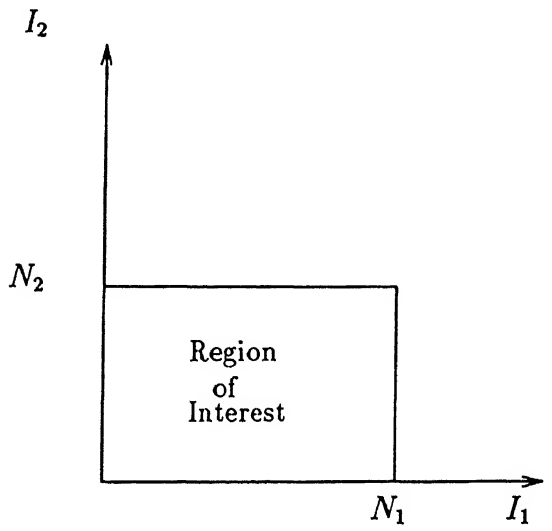


Figure 3.5: Rectangular iteration space

```
DO I1 = 1, N1
   DO I2 = I1+1, N2
      .
      .
   ENDDO
ENDDO
```

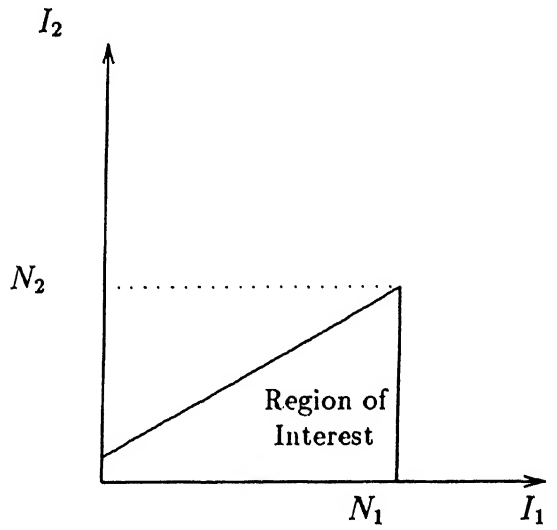


Figure 3.6: Trapezoidal iteration space

3.2.3.3 Bounds in rectangles

Consider a linear diophantine equation,

$$f(x) \equiv a_1x_1 + a_2x_2 + \ldots + a_nx_n = c$$

If $\Re$ is a rectangle, and $p_k \leq x_k \leq q_k$ for $1 \leq k \leq n$, then the minimum value is given by

$$b_{low}(f_r, \Re) = \sum_{k=1}^{n}(a_k^+ p_k - a_k^- q_k)$$

and the maximum value by

$$b_{up}(f_r, \Re) = \sum_{k=1}^{n}(a_k^+ q_k - a_k^- p_k)$$

3.2.3.4 Bounds in Trapezoids

Consider a linear diophantine equation:

$$f(x) \equiv a_1x_1 + a_2x_2 + \ldots + a_nx_n = c$$

Let the constraints for $\Re$ be:

$$p_{10} \leq x_1 \leq q_{10}$$

$$p_{20} + p_{21}x_1 \leq x_2 \leq q_{20} + q_{21}x_1$$

$$\vdots$$

$$p_{n0} + p_{n1}x_1 + \ldots + p_{n,n-1}x_{n-1} \leq x_n \leq q_{n0} + q_{n1}x_1 + \ldots + q_{n,n-1}x_{n-1}$$

Algorithm 3.3 in Figure 3.7 finds the bounds b_{low} and b_{up} for f in $\Re$.

Algorithm 3.3: Bounds in trapezoids

1. $b_{low} \leftarrow 0$
   $b_{up} \leftarrow 0$
   $(d_1, d_2, \ldots, d_n) \leftarrow (a_1, a_2, \ldots, a_n)$
   $(e_1, e_2, \ldots, e_n) \leftarrow (a_1, a_2, \ldots, a_n)$

2. for k $\leftarrow$ n *downto* 1 do
   {
   $b_{low} \leftarrow b_{low} + d_k^+ p_{k0} - d_k^- q_{k0}$
   $b_{up} \leftarrow b_{up} + e_k^+ q_{k0} - e_k^- p_{k0}$

   if $(k > 1)$
   {
   $(d_1, d_2, \ldots, d_{k-1}) \leftarrow (d_1 + d_k^+ p_{k1} - d_k^- q_{k1},$
   $d_2 + d_k^+ p_{k2} - d_k^- q_{k2}, \ldots,$
   $d_{k-1} + d_k^+ p_{k,k-1} - d_k^- q_{k,k-1})$

   $(e_1, e_2, \ldots, e_{k-1}) \leftarrow (e_1 + e_k^+ q_{k1} - e_k^- p_{k1},$
   $e_2 + e_k^+ q_{k2} - e_k^- p_{k2}, \ldots,$
   $e_{k-1} + e_k^+ q_{k,k-1} - e_k^- p_{k,k-1})$
   }
   }

3. Stop

Figure 3.7: Algorithm for finding bounds in trapezoids

3.2.4 Conclusions

After giving a fairly formal treatment to the problem of data dependence analysis, we will try to draw some conclusions about the tests themselves:

- The *GCD test* does not restrict the existence of a solution to any specific region, and searches in the entire integer space.
- Banerjee's test takes the region of interest (iteration space) into account but fails to distinguish the cases when a real solution exists in the region but an integer solution does not exist
- Both the tests are conclusive when they indicate that there is no solution and both of them never conclusively indicate that a solution exists; they only indicate that a solution *may* exist.

Taking these factors into account, a lot of research work is going on in this area.

3.2.5 Implementation Notes

As of now, only GCD and Banerjee's tests have been incorporated into the F90 compiler. The dependences between statements, if found to be existent, are recorded as data dependence edges in the Program Dependence Graph. Along with the type of dependence, the depth of the dependence and the array references causing the dependences are also stored in the edge structure.

Chapter 4

Code Restructuring

Program Restructuring encompasses all transformations affected on the program, with the objective of making it suitable for "faster" execution on a particular machine architecture. These transformations include loop interchange, node splitting, loop spreading, loop fusion, statement reordering, strip mining, cycle shrinking [23] and vector/parallel code generation. The scope of these transformations may include changing the code or the order of the code executed. However, the dependences in the original code are preserved. Therefore, if there are two executions of statements that reference a common memory location before restructuring, there will be two executions that reference the same location after the transformation. In this chapter, we will be looking at some of these restructuring techniques.

4.1 Loop Interchange

By now, the reader will have grasped the fact that programs written in a sequential language for a scalar machine are often not well suited to fully utilize vector hardware. Consider, for instance, Fortran code to compute the product of two matrices A and B [5]. The product matrix C is assumed to be initialized to zero.

```
DO J = 1, N
    DO I = 1, N
        DO K = 1, N
S1:         C( I, J ) = C( I, J ) + A( I, K ) * B( K, J )
        ENDDO
    ENDDO
ENDDO
```

On a scalar machine, this code can make optimal use of registers. The outer two loops select an element of the product matrix C; the innermost loop then performs all computations involving that element. Accessing memory repeatedly can be avoided by accumulating the intermediate values of C in a register. Thus, a good optimizing compiler can generate excellent code for this segment.

If we consider converting this code segment to vector code however, we face problems. There is a *self dependence* involving S_1 carried by the K loop which is at level 3 in the loop nest (*true dependence* at depth 3). The parallelism index of S_1 is zero and hence it cannot be vectorized.

Now, if we interchange the loops so that the K loop occupies the outermost position,

```
DO K = 1, N
    DO J = 1, N
        DO I = 1, N
S1:         C( I, J ) = C( I, J ) + A( I, K ) * B( K, J )
        ENDDO
    ENDDO
ENDDO
```

the innermost loops can be trivially vectorized. Note that interchanging loops in this case does not change the semantics of the program. The corresponding program using Fortran 90 vector operations would then be

```
DO K = 1, N
      C( 1 : N : 1, 1 : N : 1) = C( 1 : N : 1, 1 : N : 1) +
                                 A( 1 : N : 1, K ) * B( K, 1 : N : 1)
ENDDO
```

It is easy to see that this transformation permits better utilization of the vector hardware. However, effectively employing loop interchange is not a straightforward matter; it requires a study of the data dependences in the program. This section details the concepts relating to identifying interchange preventing dependences and performing profitable loop interchanges.

4.1.1 Interchange Prevention

Loop interchange falls under the category *Reordering Transformations* [4] of general code restructuring. These reordering transformations have the important characteristic that they do not change the code that is executed; rather they only affect the order in which the code is executed. More specifically, loop interchange only alters the order in which the loops in a loop nest are executed and *does not* change the code in the loop nest.

Loop interchange can be applied on a pair of loops L and L' only under the following conditions [25]:

1. Loops L and L' must be tightly nested; i.e., L surrounds L', but contains no other executable statements.
2. Loop limits of L' are invariant in L.

3. No *interchange preventing dependences* exist among the statements contained in L and L'.

Before we see *what* dependences are interchange preventing, we make some important observations. The corresponding theorems and proofs can be found in [4].

Observation 4.1 *If S_2 has a loop independent dependence upon S_1, it will be preserved by loop interchange.* □

Observation 4.2 *Any loop interchange which does not alter loops 1 through k preserves any level k dependence.* □

4.1.1.1 Interchange Preventing Dependences

A dependence that will be reversed by loop interchange is *interchange preventing* [15]. Because loop interchange corresponds to a very obvious mapping on direction vectors, interchange preventing dependences are very easy to detect when represented by direction vectors. A dependence prevents the interchange of two loops if the leftmost non "=" entry in its direction vector is "<" before interchange and ">" after interchange.

Figure 4.1[4, 5], illustrates the type of dependences that can prevent loop interchange. It depicts the possible dependences of statement S on itself during execution of the following loops:

```
DO I = 1, N1
    DO J = 1, N2
        S
    ENDDO
ENDDO
```

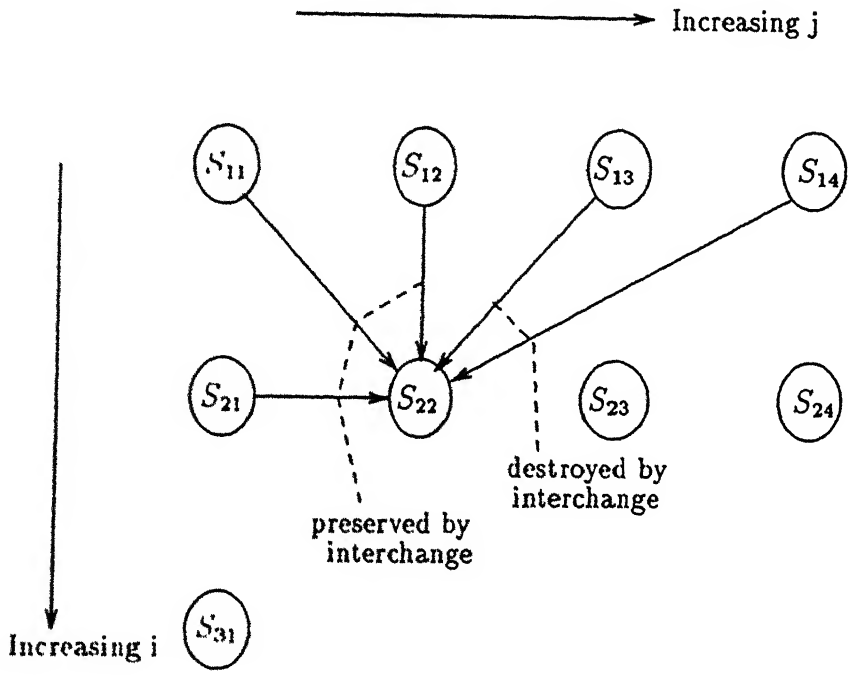


Figure 4.1: Dependence patterns in loops

Each node in the array represents one execution of statement S. S_{11} is the execution of S when both I and J are 1; S_{12} is the execution of S when I is 1 and J is 2 and so on.

Consider the statements on which S_{22} depends. If the loops are interchanged (corresponding to a transposition of the matrix) , the dependences of S_{22} on S_{11}, S_{12} and S_{21} will not be reversed, because S_{22} will still be executed after these statements in the transformed code. However, the dependence of S_{22} on S_{13} will be reversed because S_{22} will be executed before S_{13} in the interchanged code.

The reader should note that interchange preventing dependences only inhibit the interchange of certain loops. For example, a dependence described by the direction vector (<, >, <) prevents interchange of loops 1 and 2 but not loops 1 and 3 or loops 2 and 3. Further, *Observation 4.2* absolves us from worrying about loop independent dependences as they can *never* prevent the interchange of *any* loops.

Trivial extension of Figure 4.1 to the case of a DO loop body consisting of

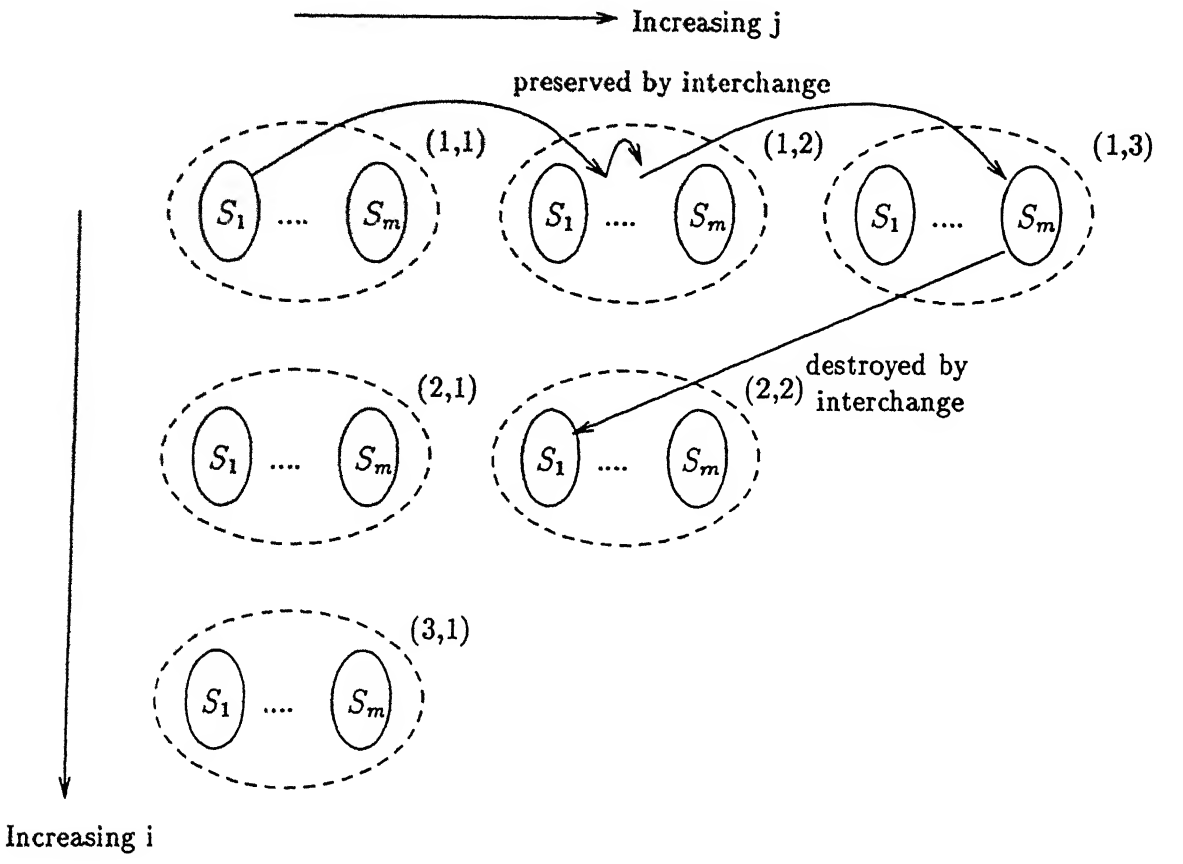


Figure 4.2: Dependence patterns among a sequence of statements

more than one statement helps in the generalization of the concept. Figure 4.2 depicts dependence patterns for the loop body:

DO I = 1, N_1
  DO J = 1, N_2
    S_1
    S_2
    S_3
    ⋮
    S_m
  ENDDO
ENDDO

In the above example, we have considered the two loops which are candidates for interchange. In Figure 4.2, the loop body S consists of a sequence of statements $S_1, S_2, S_3, \ldots, S_m$. The superscript on each oval node indicates the values of I and J during that execution. It is easy to see from Figure 4.2 that *certain* loop carried dependences from one statement to any other statement (including itself) in the loop body *may* inhibit loop interchange.

4.1.1.2 Testing for Interchange Prevention

[15, 4] use the notation γ_p^k to denote a dependence between statements S_1 and S_2 which prevents the interchange of loops p and k. There are two approaches to find out if $S_1 \gamma_p^k S_2$. The first approach is the extension of Banerjee's inequality by [15] which tests for interchange preventing dependences between adjacent loops, stated as Theorem 4.1 below. The second and less costly method involves direction vectors.

Theorem 4.1 *If S_1 and S_2 are of the form*

$$S_1 : X(f(x_1, x_2, \ldots, x_{n_1})) = \ldots$$

$$S_2 : \ldots = X(g(x_1, x_2, \ldots, x_{n_2}))$$

and occur together in at least k+1 common loops, and f and g are linear functions of the loop induction variables, i.e.,

$$f(i_1, i_2, \ldots, i_{n_1}) = a_0 + \sum_{j=1}^{n_1} a_j i_j$$

$$g(i_1, i_2, \ldots, i_{n_2}) = b_0 + \sum_{j=1}^{n_2} b_j i_j$$

then $S_1 \gamma_{k+1}^k S_2$ only if,

	Safe	Unsafe
	$k \cdots p$	$k \cdots p$
Type 1	$< \ * \ <$	$< \ * \ >$
Type 2	$<=^+ < \ * \ =$	$= \ \leq^+ \ >$
Type 3	$< \ \leq^+ \ \leq$	$<=^+ > \ * \ =$

Table 4.1: Safe and Unsafe loop interchanges

$$a_{k+1} - b_k - \sum_{i=1}^{k-1}(a_i - b_i)^-(N_i - 1) - (a_k^- + b_k)^+(N_k - 2)$$
$$-(a_{k+1} - b_{k+1}^+)^-(N_{k+1} - 2) - \sum_{i=k+1}^{n_1} a_i^-(M_i - 1) - \sum_{i=k+1}^{n_2} b_i^+(N_i - 1)$$
$$\leq \sum_{i=0}^{n_2} b_i - \sum_{i=0}^{n_1} a_i \leq$$
$$a_{k+1} - b_k + \sum_{i=1}^{k-1}(a_i - b_i)^+(N_i - 1) + (a_k^+ - b_k)^+(N_k - 2)$$
$$+(a_{k+1} + b_{k+1}^-)^+(N_{k+1} - 2) + \sum_{i=k+1}^{n_1} a_i^+(M_i - 1) + \sum_{i=k+1}^{n_2} b_i^-(N_i - 1)$$

□

Theorem 4.1 in effect determines the absence of direction vectors of the form $(= \cdots, <, >, *)$ and can be trivially extended to test for the interchange of any loops k and p. This is done by checking for feasibility of moving loop k all the way up to p by swapping adjacent loops, and then moving loop p all the way down to k.

Testing for interchange prevention gets simplified when direction vectors are used. Table 4.1 [4] lists direction vectors that permit and prohibit interchange of two loops k and p. The three types of dependences depicted in the table represent :

Type 1: A level k dependence which directly prevents interchange of loops k and p

Type 2: A level q dependence which directly prevents interchange of loops k and p, where $q \in \{ l, m, \ldots, o \}$

Type 3: A level k dependence which will reverse on interchange because of some loop $q \in \{ l, m, \ldots, o \}$.

4.1.2 Profitability of interchange

From the definition of parallelism index of a statement S, we can derive that loop interchange is profitable whenever it moves a recurrence outward, thereby decreasing $\eta(S, S)$ and increasing the parallelism index. In terms of directions of dependence, a loop interchange may be profitable when an inner "<" dependence causing loop is swapped with an outer "=" dependence causing loop. However, it is possible that the same outer loop may be having direction "<" for another dependence. When interchange is performed, this dependence edge moves inwards, with the result that its nesting level increases[1]. So, we can say that an interchange is profitable when, for a majority of statements in the loop body, the recurrence moves outwards.

4.1.3 Interchange effects on dependences.

Studying the effects of interchange is as important as the study of its feasibility. This becomes evident as one realizes that when we change the order of execution of loops as during interchange, we are actually changing the relative positions (levels) of the loops in the loop nest. The fact that loop carried dependences within loops are represented as being "carried" by particular levels further endorses a thorough analysis of the effects of interchange on the dependences. The following theorems [4] throw light upon this topic. The reader is referred to [4] for the proofs of these theorems.

Theorem 4.2 *If loops k and p ($k < p$) are validly interchanged, then all loop independent dependences, all dependences with level $< k$, and all dependences with level $> p$ are unaffected by the interchange.* □

[1]Such dependences are called *interchange sensitive* dependences

Theorem 4.3 *If loops k and p ($k < p$) are validly interchanged, then all level p dependences in the original code become level k dependences in the transformed code.* □

Theorem 4.4 *If loops k and p ($k < p$) are validly interchanged, then all level k dependences in the original code become level x dependences in the transformed code, where $k \leq x \leq p$.* □

Theorem 4.5 *In a valid loop interchange of loops k and p, a level x edge will either become a level k edge or remain a level x edge, where $k < x < p$.* □

4.1.4 Two approaches to interchanging loops

[4] describes a modified version of the parallel code generation algorithm (codegen [15]) to incorporate loop interchange. We will, on the other hand be concentrating on manipulating the PDG before the already implemented procedure codegen is called in order to minimize changes to it. This subsection describes two approaches to the task. Algorithm 4.1 is applicable to interchanging adjacent loops. For interchanging non-adjacent loops, the dependences within the loop nest have to be augmented with the addition of direction vectors. Algorithm 4.2 in Figure 4.4 outlines the resulting interchange procedure.

While the first approach involves nothing more than the straightforward application of the concepts presented so far, the main reason why it cannot be applied to interchanging non-adjacent loops is the cost of Theorem 4.1 in this case. In other words, if there are r loops between arbitrary candidates loops for interchange k and p ($k < p$), $2r+1$ inequalities similar to γ of Theorem 4.1 must be tested to guarantee the safety of loop interchange. $r+1$ tests correspond to moving the p loop out to the position of the k loop, and r tests to moving the k loop to the original position of p, all by adjacent interchanges.

This bottleneck for interchange can be eliminated by using direction vectors to test for feasibility of interchange. Algorithm 4.2 in Figure 4.4 outlines the

```
Algorithm 4.1
Step 1.
    During dependence analysis,
        for (each level = MaxDepth downto 1){
            (1.1) Apply Theorem 4.1 and mark dependence
                edges to indicate whether
                they prevent interchange of loops level and level − 1
            (1.2) If (no interchange preventing dependences are found){
                Swap the elements corresponding to level and level − 1
                being considered for interchange;
                Test for dependence;
                If (dependence has disappeared at that level){
                    add level to the list of profitable interchanges
                    for the dependence under consideration;
                }
            }
        }

Step 2.
    for (each loop nest)
        for (each dependence within)
            for (each level in the list of profitable interchanges){
                if (no interchange preventing dependences exist at that level){
                    switch the loops level and level − 1;
                    /* Update dependences */
                    for (each dependence in the loop nest)
                        if (depth of dependence == level)
                            depth of dependence = level − 1;
                        }
                    }
                }
            }
```

Figure 4.3: Algorithm for interchange of adjacent loops

resulting approach, assuming that the dependence direction vectors have been found during dependence analysis.

4.2 Node Splitting

It is fairly common to find recurrences that have an essential dependence edge that represents an output dependence or antidependence. Single statement recurrences due to these "pseudo" dependences can be safely ignored thanks to the fetch-before-store vector semantics of Fortran 90. Matters are not as simple as this when two statements are involved in a dependence cycle, where one of the edges is an antidependence edge. This section deals with the issues involved in breaking such dependence cycles.

Node Splitting involves renaming the variable or array reference that causes the antidependence edge, and combined with Statement Reordering, forms a powerful vectorizing aid. While breaking recurrences due to true dependences is complicated and in many cases not possible, Node Splitting is fairly straight forward. The technique is best illustrated with the aid of the example shown in Figure 4.5[5]. F() and G() are some functions.

The dependence graph for the example is shown next to it. As it is, it is not possible to vectorize the loop block in Figure 4.5 because of the dependence cycle. If we introduce a new variable and assign to it the variable causing the antidependence as in Figure 4.6 , the dependence cycle is broken. Now we can safely perform a topological sort on the dependence graph (*statement reordering*) to get the vectorizable code segment in Figure 4.7. General prudence however, dictates that one has to cautiously weigh the efficiency gained by this transformation against additional storage requirements for the temporary arrays.

Algorithm 4.2. *Interchange(LoopNest)*

```
/* SCCs in LoopNest are found and stored in scc_components */
find_sccs_in_dependence_graph( LoopNest, scc_components );
distribute_loops(scc_components);
/* Consider loops of farthest distance first and proceed towards
     interchanging adjacent loops; this automatically generates the
     best sequence for interchange */
for (each scc_component[i], 1 ≤ i ≤ no_of_components){
    for ( k = 1; k < max_levels; k++ ) {
        for ( p = max_levels; p > k; p-- ) {
            /* Consider interchanging loops k and p */
            /* The SCCs at level p in the region scc_component[i] are found.
            N1 is used to store the number of cycle less regions that result
            (statements vectorizable) */
            find_sccs_at_level_p (scc_component[i], N1);
            /* Is safe interchange possible ? if no, go to next p */
            if (direction_vector[p] == ">" in any dependence edge)
                continue;
            /* Apply rules of table 4.1 to find out the existence of
            interchange preventing dependences for loops k and p */
            find_if_interchangeable(k, p)
            if (k and p are interchangeable) {
                /* The SCCs at level k in the region scc_component[i] are found.
                N2 is used to store the number of cycle less regions that
                result (statements vectorizable) */
                find_sccs_at_level_k (scc_component[i], N2);
                if (N1 < N2) {
                    swap_loops(k, p);
                    /* Use direction vectors to identify new "carriers"
                    of dependences within scc_component */
                    update_dependences(scc_component[i]);
                    /* go to next k */
                    break;
                }
            }
        }
    }
}
```

Figure 4.4: Algorithm for general loop interchange

```
DO I = 1, N
S1: A ( I ) = G ( X ( I + 1 ) + X ( I ) )
S2: X ( I + 1 ) = F ( B ( I ) )
ENDDO
```

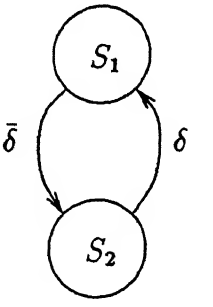


Figure 4.5: Sample program for Node Splitting

```
DO I = 1, N
S': TEMP ( I ) = X ( I + 1 )
S1: A ( I ) = G ( TEMP ( I ) + X ( I ) )
S2: X ( I + 1 ) = F ( B ( I ) )
ENDDO
```

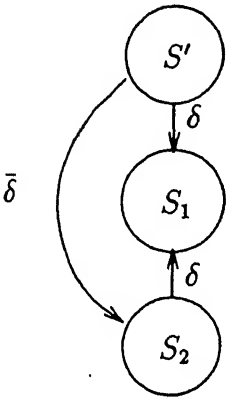


Figure 4.6: Node S_1 has been split

4.2.0.1 Implementation Notes

During data dependence analysis, array references causing dependences are stored in each edge structure. This facilitates quick identification of the array reference causing the anti dependence edge and replacing it by the new variable.

4.3 Vector Code Generation

The vector code generation algorithm used in the F90 compiler, in principle works on the same lines as the one in [5].

Coarsely, the code generation technique consists of the following steps.

Step 1: Find all the *strongly connected regions* in the program dependence graph consisting of *both* the control dependence and data dependence edges

```
DO I = 1, N
S': TEMP ( I ) = X ( I + 1 )
S2: X ( I + 1 ) = F ( B ( I ) )
S1: A ( I ) = G ( TEMP ( I ) + X ( I ) )
ENDDO
```

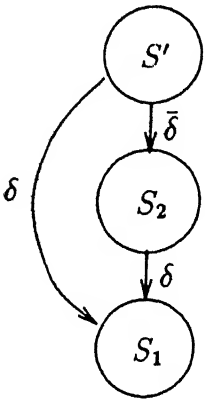


Figure 4.7: Node Splitting followed by Reordering

Step 2: Reduce the dependence graph to an acyclic graph by treating each strongly connected region as a single node, or π-block

Step 3: Generate code for each π-block in an order consistent with the dependences. Topological sort is performed, to first generate code for blocks that depend on no others, then for blocks that depend only on blocks for which already code has been generated, etc.

The codegen procedure of [5] operates on the data dependence graph and vectorizes statements not contained in any strongly connected component of data dependences. To handle conditional statements, *IF-conversion*[3] is first performed converting control dependences to data dependences. However, the PDG provides a unified way of representing both the control and data dependences and allows treating both the types of edges uniformly. Any node in the PDG not contained in a strongly connected component consisting of *both* the control and data dependences can be vectorized. Thus, the essential difference lies in the way the π-blocks are found. The procedure vectcodegen given as Algorithm 4.3 in Figure 4.8 captures the essence of vectorization, as used in the F90 vectorizer.

4.4 Vectorizing Conditionals

Vector languages such as Fortran 90 include features that allow conditional assignments to take place simultaneously on an array. Also called masked array

```
Algorithm 4.3:vectcodegen(Region, Pdg, MinNestLevel)
    /* Region is the region for which vector code must be generated */
    /* Pdg is the Program dependence graph of the statements in Region */
    /* MinNestLevel is the minimum nesting level of possible parallel loops */
π_blocks = FindSCCsInPDG ( Region, Pdg );
for ( each π_block[i], 1 ≤ i ≤ NoOfComponents ){
    if ( π_block[i] is strongly connected ){
        generate a level MinNestLevel DO statement;
        Region_i = PDG considering all edges in Pdg internal
        to π_block[i] with level MinNestLevel+1 or greater;
        vectcodegen( π_block[i], MinNestLevel+1, Region_i );
        generate level MinNestLevel ENDDO;
    else
        NoEnclosingLoops = FindNoOfEnclosingLoops ( π_block[i] );
        generate vector statements for π_block[i] in
        NoEnclosingLoops - MinNestLevel + 1 dimensions;
    }
}
```

Figure 4.8: Algorithm for Vector code generation

assignments, these are used to mask the evaluation of expressions and assignment of values in array assignment statements, according to the value of a logical expression. Fortran 90 provides this facility in the form of WHERE construct or WHERE statement. The syntax of WHERE statement and construct is produced here for reference [11].

where-stmt	IS	WHERE (*mask-expr*) *assignment-stmt*
where-construct	IS	*where-construct-stmt*
		[*assignment-stmt*]
		elsewhere-statement
		[*assignment-stmt*]...]
		end-where-stmt
where-construct-stmt	IS	WHERE (*mask-expr*)
mask-expr	IS	*logical-expr*
elsewhere-stmt	IS	ELSEWHERE
end-where-stmt	IS	END WHERE

The following subsection discusses the issues involved in converting IF statements to WHERE form.

4.4.0.2 Issues

When we talk about IF statements, we are talking about conditional execution and hence are firmly in the domain of control dependences. Before the program dependence graph was proposed, a process called "IF-conversion" [3] was used to convert control dependences to data dependences.

IF-conversion is accomplished by converting all statements under the control of an IF or branch to conditional statements. These conditional statements are then translated to vector statements viewing the conditional expression as just another "read" in (input to) the statement.

However, IF-conversion is non-trivial and hopelessly fragments the control structure of a program. For instance, the DO loop nest

```
        DO I = 1, 100
             IF ( A(I) > 100 ) goto 60
S1:               A(I) = A(I) * 5
                  IF ( B(I) > 100 ) goto 80
S2:                    B(I) = B(I) - 100
S3: 60       A(I) = A(I) - B(I)
S4: 80       B(I) = A(I) + 5
     ENDDO
```

upon IF-conversion translates to,

```
        DO I = 1, 100
             BR1 = A(I) > 100
S1:          IF ( .NOT. BR1 ) A(I) = A(I) * 5
             IF ( .NOT. BR1 ) BR2 = B(I) > 100
S2 :         IF ( .NOT. BR1 .AND. .NOT. BR2 ) B(I) = B(I) - 100
S3 :         IF ( BR1 .OR. .NOT. BR2 ) A(I) = A(I) - B(I)
S4:          B(I) = A(I) + 5
     ENDDO
```

This can be converted (fortunately!) to a sequence of WHERE statements. Quite often, IF statements can be part of recurrences, in which case vectorization is not possible. It is under such conditions that recovering the original control flow of the program from the IF-converted one becomes necessary.

In the cases where the PDG is used to represent the program, as in our case, there is no such fragmentation of the control flow. As the PDG allows uniform handling of control and data dependences, while finding the strongly connected components, we consider *both* the control and data dependence edges and handle

IF statements just like the ordinary assignment statements.

4.4.0.3 Loop Exit IFs

The class of IF statements within DO loops that conditionally branch out of the loop, are called *loop exit IFs*. There is no general procedure to vectorize loop exit IFs and some heuristics may be applied [25]. There are several issues to be considered:

- The resulting vectorized code may take longer to execute than the original serial code. For example,

```
      DO I = 1, N
           A(I) = B(I) + C(I)
           IF ( A(I) > REF(I) ) goto label
      ENDDO
label: ...
```

If simplistically vectorized, the above segment results in

```
S1:  A(1:N) = B(1:N) + C(1:N)
S2:  FLAG(1:N) = A(1:N) > REF(1:N)
S3:  I = FIRST_ONE ( FLAG(1:N) )
S4:  IF ( I > 0 ) goto label
label: ...
```

FIRST_ONE is a function that returns the index value of the first array element set to 1. If N is large, and the loop exit condition is satisfied for some small I, then the serial version will execute only I iterations while the vectorized version will execute all iterations of S_1 and S_2 before finding that the last N−I iterations were indeed, unnecessary.

- Observe that, in the above segment, all the values of A in S_1 are changed so that the vectorized code, when executed may produce incorrect results. This can be circumvented by assigning all results computed in the loop to some compiler temporary arrays. Proper results can be copied to the program variables after the loop exit condition is found.
- It is difficult to predict the effect of executing the vectorized code. For instance, suppose that for some computation of S_1 a run time error (say overflow) is bound to occur. Also suppose that this instance of S_1 is not executed in the sequential version; all goes well. In the vector version however, as all instances of S_1 are executed, there will be a fault.

In general, it is extremely difficult to vectorize loop exit IFs as most of these loop conditions are determined at run time. Predicting their run time behavior at compile time is impossible with the existing compiler techniques.

Chapter 5

Epilogue

5.1 Summary

The vectorizer is capable of optimizing most of the practically occurring code patterns. In this thesis, we have concentrated on describing concepts related to loop optimizations for maximal extraction of concurrency in code segments. While all the other phases of the compiler use the syntax tree as intermediate representation, the vectorizer uses the Program Dependence Graph as the internal representation. The increase in time taken by the compiler because of the conversion from the syntax tree to the PDG is offset by the convenience of handling control and data dependences uniformly. Array dependence analysis forms the backbone of vectorization. Going deeper with array subscript analysis rather than the cursory examination based on array names goes a long way in uncovering the (possible) absence of dependences. Loop interchange is performed on a loop nest only if it results in increasing the parallelism index of the majority of the statements within the loop nest. So, if loop interchange does take place, it invariably results in producing a *more* vectorized code than was possible before. While Node Splitting breaks dependence cycles with antidependences as components, the vector code generation routine recursively tries to break true dependence cycles at greater depths. The presence of a vector facility for conditional assignment as an added bonus is tapped to convert IF

constructs that do not branch out of loops to equivalent WHERE constructs. Finally, we have discussed issues relating to vectorizing the loop-exit IFs.

5.2 Status of Implementation

Currently, the F90 vectorizer is capable of performing the following loop optimizations:

- loop normalization,
- auxiliary induction variable elimination,
- interchange of adjacent loops,
- node splitting, and
- generation of vector code for array assignments and IF constructs that do not branch out of the loop.

The dependence analysis phase uses GCD and Banerjee's tests to determine dependences between array references.

5.3 Testing of the software

The vectorizer is an optional feature of the compiler that can be invoked by specifying the "-V" option on the command line. Examples were designed to test each phase of the vectorizer. More specifically, examples to test various scenarios in the following were conceived and used:

- Performing *loop normalization*
- Identification and elimination of *auxiliary induction variables*
- Performing *array dependence analysis*
- Performing feasible and profitable *loop interchanges*
- Performing *node splitting*

- Vectorizing *conditionals,* and finally,
- Generating *vector code*

The vectorizer, being a part of the larger compiler project consisting of phases like the *front-end* and the *scalar optimizer* coming before, and the unparser coming after, has to work hand-in-hand with them. So test cases were crafted to validate the *integration* of the vectorizer with the compiler. About 40 programs of size ranging from 5 lines to 35 lines of code were used to test the vectorizer.

5.4 Future Directions

Even though the F90 compiler extracts parallelism to generate the vectorized code in most of the practical cases, there is scope for improvement. In particular, future work may concentrate on the following areas:

- Augmenting the dependence testing by the addition of more dependence tests. While the GCD test proves ineffective in most of the practical cases, Banerjee's test requires the bounds of the region to be known. Both of these tests are *inexact tests* and incorporating some more tests like the I-test[17], Power test[26], Delta test[14], etc., will help in somewhat neutralizing the inaccuracies due to GCD and Banerjee's tests.
- Handling input instructions. As of now, the vectorizer cannot identify dependences resulting from the presence of such statements in loop bodies.
- Another direction in which the dependence testing phase may be extended is in handling procedures and functions. This will require analysis to be done at the local and the global levels to identify dependences between references under the same scope.
- The vectorizer as of now handles only adjacent loop interchanges. The reason for this decision was the expense involved in computing the feasibility and profitability of general loop interchange. However, if need be,

this phase can be extended to perform general loop interchanging using the algorithm suggested in this thesis.

- Presence of *loop exit IFs* in DO loops severely hampers vectorization. This is so because in most of the cases, the conditions under which the loop is exit become evident only during execution. However, in some specific cases, these IFs are *less* unpredictable and can possibly be vectorized. A study can be done to enumerate conditions under which such vectorization can be carried out, and with the application of some *ad-hoc* techniques, concurrency can be tapped.
- Extensions to the project can also concentrate on determining the possible existence of program segments equivalent to REPEAT-UNTIL loops:

```
label:  ...
        ⋮
        IF ( condition ) goto label;
        ...
```

and WHILE-DO loops:

```
label1: IF ( condition ) goto label2;
        ⋮
        goto label1;
        ⋮
label2: ...
```

Bibliography

[1] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *The Design and Analysis of Computer Algorithms,* Addison-Wesley, 1980.

[2] Aho, A.V., Sethi, R. and Ullman, J.D., *Compilers: Principles, Techniques and Tools.*, Addison-Wesley, 1986.

[3] Allen, R., Kennedy, K., Porterfield, C., and Warren,J. *Conversion of Control Dependences to Data Dependences,* Proceedings of the 10^{th} Annual ACM Symposium on Principles of Programming Languages, January 1983.

[4] Allen, R. and Kennedy, K. *Automatic Loop Interchange*. Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction, Vol 19, No. 6, June 1984.

[5] Allen, R. and Kennedy, K. *Automatic Translation of FORTRAN programs to Vector Form.* ACM Transactions on Programming Languages and Systems, Vol 9, No. 4, October 1987, pages 491-542.

[6] Allen, J.R., Callahan, D. and Kennedy, K. *Automatic Decomposition of Scientific programs for Parallel Execution.* Proceedings of the 14^{th} Annual ACM Symposium on Principles of Programming Languages, January 1987, pages 63-76.

[7] Banerjee, U., *Dependence Analysis for Supercomputers,* Kluwer Academic, 1988.

[8] Baxter, W. and Bauer, III, H.R., *The Program dependence Graph and Vectorization,* Proceedings of the 16^{th} Annual ACM Symposium on the Principles of Programming Languages, January 1989, pages 1-11.

[9] Counihan, *Fortran 90,* Pitman, 1991.

[10] Ferrante, J., Ottenstein, K.J., and Warren, J.D. *The Program Dependence Graph and its use in Optimizations,* ACM Transactions on Programming Languages and Systems, Vol 9, No. 3, October 1987, pages 319-349.

[11] Adams, C.J., Brainend, W.S., Martin, J.T., Smith, B.T. and Wagener, J.L., *Fortran 90 Handbook, Complete ANSI/ISO Reference,* McGraw-Hill Book Company, 1992.

[12] Fortran 8x, ACM SIGPLAN Special Interest Publication on Fortran, ACM Press, May 1989.

[13] Lengauer, T. and Tarjan, R.E. *A fast algorithm for finding dominators in a flowgraph,* ACM Transactions on Programming Languages and Systems, July 1979, pages 121-141.

[14] Li, Z., Yew, P.C., and Zhu, C.Q., *An efficient data dependence analysis for parallelizing compilers,* IEEE Transactions on Parallel and Disptributed Systems, Vol. 1, No. 1, January 1990.

[15] Kennedy, K., *Automatic Translation of Fortran Programs to Vector Form,* Rice Technical Report 476-029-4, Rice University, October 1980.

[16] Kennedy, K., *Compiler Technology for Machine Independent Parallel Programming,* International Journal of Parallel Programming, Vol 22, No. 1, February 1994, pages 79-98.

[17] Kong, X., Klappholz, D., and Psarris, K., *The I test:A new test for subscript data dependence,* International Conference on Parallel Processing, 1990.

[18] Kuck, D.J., Kuhn, R.H., Leasure, B. and Wolfe, M., *The Structure of an Advanced Vectorizer for Pipelined Processors,* Proceedings of COMPSAC 80, the 4^{th} International Computer Software and Applications Conference, Ocober 1980, pages 709-715.

[19] Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B. and Wolfe, M., *Dependence Graphs and Compiler optimizations,* Conference Record of the 8^{th}

Annual ACM Symposium on the Principles of Programming Languages, January 1981, pages 207-218.

[20] Murali, B., *Optimization of Fortran 90 Programs : Alias Analysis,* M.Tech Thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, March 1994.

[21] Narasimhan, G., *Code Generator for Convex C220,* M.Tech Thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, October 1994.

[22] Pinter, S.S., and Pinter, R.Y., *Program Optimization and Parallelization using Idioms,* ACM Transactions on Programming Languages and Systems, May 1994, pages 305-327.

[23] Polychronopoulos, C.D., *Parallel Programming and Compilers,* Kluwer Academic, 1988.

[24] Singh, R. and Purohit, V.D., *A Tool for Vectorizing Sequential Programs,* B.Tech. Project Report, Department of Computer Science and Engineering, IIT Kanpur, April 1991.

[25] Wolfe, M., *Optimizing Supercompilers for Supercomputers,* MIT Press, 1989.

[26] Wolfe, M. and Tseng, C.W., *The power test for data dependence,* IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 5, September 1992.